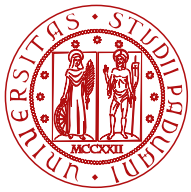


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in

Ingegneria Informatica

Studio e implementazione di density clustering DBSCAN approssimato

Relatore:

Ch.mo Prof. Geppino Pucci

Laureando:

Lorenzo Cazzador

Correlatore:

Ch.mo Prof. Andrea Pietracaprina

Anno Accademico 2022/2023

Data di Laurea 29/09/2023

Sommario

Il clustering è una primitiva di machine learning non supervisionato che consiste nel raggruppare oggetti simili in insiemi omogenei, chiamati cluster. Tali insiemi vengono creati in modo tale che gli oggetti appartenenti allo stesso cluster abbiano un grado di somiglianza maggiore rispetto ad oggetti presenti in altri cluster. Il DBSCAN è un metodo di clustering basato sulla densità: dato un insieme di punti in uno spazio euclideo multidimensionale, lo scopo è creare un cluster per ogni insieme di punti ad alta densità. Per dimensioni superiori a due, la complessità temporale del DBSCAN, nel caso peggiore, è provatamente superlineare, la qual cosa lo rende inadatto a gestire grandi moli di punti. Al fine di affrontare questa problematica, il presente lavoro si propone di analizzare e implementare un algoritmo di DBSCAN approssimato che migliora la complessità temporale, pur restituendo un insieme di cluster ragionevolmente assimilabili a quelli del DBSCAN esatto. L'algoritmo approssimato mira a ridurre significativamente i tempi di esecuzione, consentendo una maggiore scalabilità e adattabilità del metodo di clustering in spazi di alta dimensionialità, a scapito di una soglia quantitativa di errore che può essere resa arbitrariamente piccola.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

Capitolo 1

Introduzione

Il DBSCAN – *Density Based Spatial Clustering of Applications with Noise* – è un metodo per eseguire il clustering di punti appartenenti a uno spazio euclideo a più dimensioni. Questa tesina considera il problema del calcolo dei cluster DBSCAN utilizzando la distanza euclidea e assumendo che non ci siano indici pre-esistenti.
Per uno spazio euclideo a 2 dimensioni esiste un algoritmo che risolve il problema del DBSCAN esatto con una complessità temporale pari a $O(n \log n)$, dove n è il numero di punti del dataset, ma per dimensioni $d \geq 3$ è dimostrato che il problema richiede $\Omega(n^{4/3})$ tempo per essere risolto [6]. Per superare questo limite, consideriamo una versione approssimata del problema chiamata *ρ-approximate DBSCAN*, che nella maggior parte dei casi individua gli stessi cluster di DBSCAN, tranne se i cluster sono "instabili", ovvero, cambiano ogni qualvolta i parametri di input vengono leggermente modificati. Il problema *ρ-approximate DBSCAN* può essere risolto in un tempo atteso $O(n)$ indipendentemente dalla costante dimensionale d.

1.1 Density-based clustering

Il *density-based clustering* è uno degli argomenti principali nel data mining, utilizzato in molti campi, tra cui il riconoscimento di pattern, l'analisi di immagini, il recupero di informazioni, la bioinformatica, la compressione dei dati e l'apprendimento automatico. Si può definire il problema come segue:
Dato un insieme P di n punti nello spazio d-dimensionale $\mathbb{R}^d$, l'obiettivo è raggruppare i punti di P in sottoinsiemi – chiamati cluster – in modo tale che i cluster siano separati da

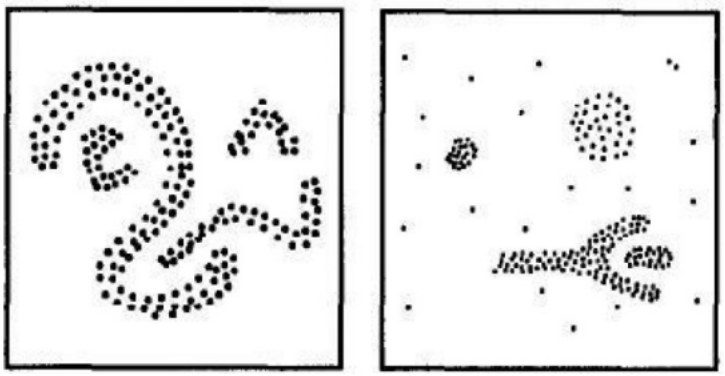


Figura 1.1: Esempi di density-based clustering tratti da Ester et al.

"regioni sparse".

La Figura 1.1 mostra due esempi tratti da Ester et al. [5]: quello a sinistra contiene quattro cluster a forma di serpente, mentre quello a destra contiene tre cluster e del rumore.

Il vantaggio principale del density-based clustering, rispetto ad altri metodi, è la capacità di individuare cluster aventi forme arbitrarie.

Il density-based clustering può essere ottenuto utilizzando diversi metodi, che differiscono principalmente nella definizione di "regione densa" e nei criteri per unire le regioni dense al fine di formare i cluster.

DBSCAN è un metodo inventato da Ester et al. [5] e definisce la "densità/scarsità" attraverso due parametri:

- ε: un valore reale positivo;
- $MinPts$: una costante intera positiva piccola.

Consideriamo $B(p,\varepsilon)$, ovvero la palla d-dimensionale di raggio ε centrata nel punto p. Tale palla è "densa" se contiene un numero di punti di P che sia maggiore o uguale al valore della costante $MinPts$.

DBSCAN forma i cluster basandosi sulla seguente logica: se $B(p,\varepsilon)$ è densa, allora tutti i punti contenuti in $B(p,\varepsilon)$ saranno aggiunti allo stesso cluster di p. Questo crea un "effetto catena": ogni volta che un nuovo punto p' con una $B(p',\varepsilon)$ densa è aggiunto al cluster di p, anche tutti i punti contenuti in $B(p',\varepsilon)$ saranno aggiunti allo stesso cluster. Il cluster di p continua a crescere in questo modo fino alla massima estensione possibile.

Capitolo 2

Definizioni preliminari

Sia P un insieme di n punti nello spazio d-dimensionale $\mathbb{R}^d$. Dati due punti $p, q \in \mathbb{R}^d$ indichiamo con $dist(p,q)$ la distanza euclidea tra p e q.

Definizione 2.1. Siano $p \in P$ e $r \in \mathbb{R}^*_+$. Indichiamo con $B(p,r)$ la palla chiusa di raggio r centrata nel punto p, definita come

$$B(p,r) = \{\, x \in P \mid dist(x,p) \leq r \,\}$$

Si ricorda che DBSCAN è definito da due parametri: ε e $MinPts$.

Definizione 2.2. Un **core point** è un punto $p \in P$ tale che $B(p,\varepsilon)$ copra almeno $MinPts$ punti di P (incluso p stesso).

Qualora p non soddisfi il requisito per essere un core point, si dice che è un *non-core point*. Supponiamo che P sia l'insieme di punti nella Figura 2.1, dove $MinPts = 4$ e i due cerchi hanno raggio ε, sono mostrati in nero i core point e in bianco i non-core point.

Definizione 2.3. Un punto $q \in P$ è **density-reachable** da $p \in P$ se esiste una sequenza di punti $p_1, p_2, \ldots, p_t \in P$ (per qualche intero $t \geq 2$) tale che

- $p_1 = p$ e $p_t = q$
- $p_1, p_2, \ldots, p_{t-1}$ sono core point
- $p_{i+1} \in B(p_i, \varepsilon)$ per ogni $i \in [1, t-1]$

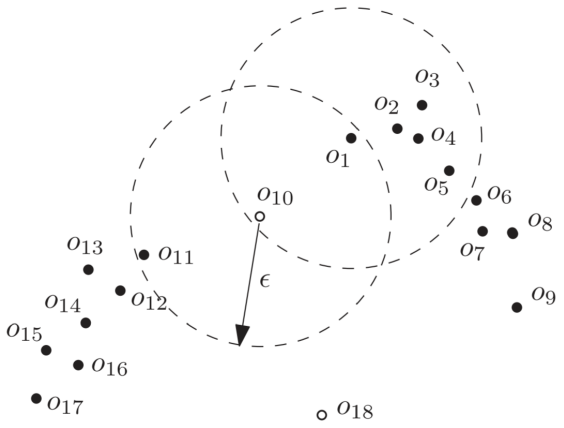


Figura 2.1: Un dataset di esempio (i due cerchi hanno raggio ε; $MinPts = 4$).

Si noti che i punti p e q possono coincidere, quindi un punto è sempre density-reachable da se stesso. Nella Figura 2.1, ad esempio, o_1 è density-reachable da se stesso; o_{10} è density-reachable da o_1 e da o_3 (attraverso la sequenza o_3, o_2, o_1, o_{10}). D'altra parte, o_{11} non è density-reachable da o_{10} (poiché o_{10} non è un core point).

Definizione 2.4. Un **cluster** C è un sottoinsieme non vuoto di P tale che

- (Maximality) Se un core point $p \in C$, allora tutti i punti density-reachable da p appartengono anch'essi a C.
- (Connectivity) Per ogni coppia di punti $p_1, p_2 \in C$, esiste un punto $p \in C$ tale che sia p_1 che p_2 siano density-reachable da p.

Tale definizione implica che ogni cluster contenga almeno un core point, ovvero p. Nella Figura 2.1, il sottoinsieme $\{o_1, o_{10}\}$ non è un cluster perché non comprende tutti i punti density-reachable da o_1. D'altra parte, il sottoinsieme $\{o_1, o_2, o_3, \ldots, o_{10}\}$ è un cluster. Ester et al. [5] hanno fornito una dimostrazione che l'insieme dei punti P ammette un unico insieme di cluster, che dà origine a

Problema 1. *Il **problema DBSCAN** consiste nel trovare l'unico insieme $\mathcal{C}$ di cluster di P.*

Dato in input l'insieme di punti P, rappresentato nella Figura 2.1, il problema dovrebbe restituire due cluster: $C_1 = \{o_1, o_2, o_3, \ldots, o_{10}\}$ e $C_2 = \{o_{10}, o_{11}, \ldots, o_{17}\}$.

Osservazione. Un cluster può contenere sia core point che non-core point. I non-core point appartenenti ad un cluster prendono il nome di *border point*. Alcuni punti potrebbero

non appartenere a nessun cluster; vengono chiamati *noise point*. Nella Figura 2.1, o_{10} è un border point, mentre o_{18} è un noise point.

I cluster in $\mathcal{C}$ non sono necessariamente disgiunti; ad esempio, nella Figura 2.1, o_{10} appartiene sia a C_1 che a C_2. In generale, i border point possono appartenere a più di un cluster, mentre un core point appartiene sempre ad un unico cluster.

Capitolo 3

Il problema DBSCAN approssimato

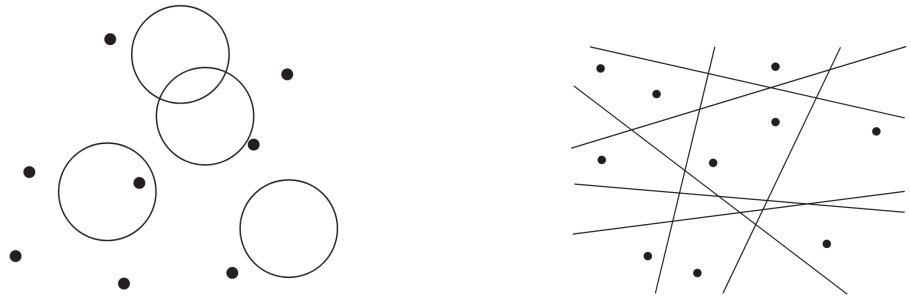
All'inizio di questo capitolo, andremo a delineare in modo rigoroso la motivazione che ha spinto all'introduzione di una versione approssimata del problema DBSCAN. Per procedere in questa direzione, cominciamo col presentare un problema affine, noto come USEC (*Unit-Spherical Emptiness Checking*), che definiamo nel seguente modo:

Problema 2. *Siano S_{pt} un insieme di punti e S_{ball} un insieme di palle aventi lo stesso raggio, entrambi appartenenti allo spazio $\mathbb{R}^d$, dove la dimensionalità d è una costante. Lo scopo del problema USEC è determinare se esiste un punto di S_{pt} che è contenuto in qualche palla dell'insieme S_{ball}.*

Ad esempio, nel caso della Figura 3.1a, la risposta è *si*.
Poniamo $n = |S_{pt}| + |S_{ball}|$. In 3 dimensioni, il problema USEC può essere risolto in $O(n^{4/3} + \log^{4/3} n)$ tempo atteso [1]. Trovare un algoritmo che risolve tale problema con una complessità temporale $o(n^{4/3})$ è un problema aperto nella geometria computazionale ed è ritenuto alquanto impossibile [4].
Quando la dimensionalità d è maggiore o uguale a 5, la risoluzione del problema USEC è strettamente correlata a quella del problema di Hopcroft, così definito:

Problema 3. *Siano S_{pt} un insieme di punti e S_{line} un insieme di rette, il tutto appartenente allo spazio $\mathbb{R}^2$. Lo scopo del problema di Hopcroft è determinare se esiste un punto di S_{pt} che interseca qualche retta dell'insieme S_{line}.*

Ad esempio, nel caso della Figura 3.1b, la risposta è *no*.



(a) Problema USEC **(b)** Problema di Hopcrof

Figura 3.1: Due problemi rilevanti di geometria.

Il problema di Hopcroft può essere risolto in un tempo leggermente superiore a $O(n^{4/3})$, dove $n = |S_{pt}| + |S_{line}|$, per una trattazione più dettagliata si può fare riferimento a Matousek [9]. Inoltre, si ritiene che $\Omega(n^{4/3})$ rappresenti un limite inferiore su quanto velocemente il problema possa essere risolto [4].

È noto che il problema di Hopcroft rappresenta un passaggio fondamentale per una vasta gamma di altri problemi [4]. Diciamo che un problema P è *Hopcroft-hard* se un algoritmo che risolve P in tempo $o(n^{4/3})$ implica un algoritmo che risolve il problema di Hopcroft in tempo $o(n^{4/3})$. In altre parole, un limite inferiore pari a $\Omega(n^{4/3})$ sul tempo per risolvere il problema di Hopcroft implica lo stesso limite inferiore per il problema P.

Ora, esaminiamo la relazione tra il problema DBSCAN e i due problemi enunciati sopra. All'interno dell'articolo pubblicato da Gan e Tao [6] è stato dimostrato il seguente teorema

Teorema 3.1. *Le seguenti affermazioni sono vere riguardo al problema DBSCAN:*

- *È Hopcroft-hard in qualsiasi dimensionalità $d \geq 5$. In altre parole, il problema richiede un tempo $\Omega(n^{4/3})$ per essere risolto, a meno che il problema di Hopcroft possa essere risolto in un tempo $o(n^{4/3})$.*

- *Per $d = 3$ e $d = 4$, il problema richiede un tempo $\Omega(n^{4/3})$ per essere risolto, a meno che il problema USEC possa essere risolto in un tempo $o(n^{4/3})$.*

Come specificato sopra, attualmente, non esiste un algoritmo che risolva il problema di Hopcroft o il problema USEC con un tempo di esecuzione $o(n^{4/3})$ – un algoritmo con tale complessità rappresenterebbe una celebre svolta nell'informatica teorica.

Il risultato di tale teorema indica la necessità di ricorrere all'approssimazione se si vuole ottenere un tempo di esecuzione quasi lineare per dimensionalità $d \geq 3$.

3.1 Definizioni

Sia P l'insieme di input costituito da n punti in $\mathbb{R}^d$. Consideriamo ancora i parametri ε e $MinPts$, ma in aggiunta anche un terzo parametro ρ, una qualsiasi costante positiva arbitrariamente piccola, che determina il grado di approssimazione.
Successivamente, riprendiamo le definizioni di base di DBSCAN enunciate nel Capitolo 2, e modifichiamo alcune di esse nelle loro "versioni ρ-approssimate". Innanzitutto, manteniamo la definizione di punto core/non-core e anche il concetto di density-reachability e introduciamo la seguente definizione:

Definizione 3.2. Un punto $q \in P$ è **ρ-approximate density-reachable** da $p \in P$ se esiste una sequenza di punti $p_1, p_2, \ldots, p_t \in P$ (per qualche intero $t \geq 2$) tale che

- $p_1 = p$ e $p_t = q$
- $p_1, p_2, \ldots, p_{t-1}$ sono core point
- $p_{i+1} \in B(p_i, \varepsilon(1+\rho))$ per ogni $i \in [1, t-1]$

Si noti la differenza tra quanto scritto sopra e la Definizione 2.3: nel terzo punto, il raggio della palla è aumentato a $\varepsilon(1+\rho)$.
Consideriamo un piccolo set di input P, mostrato nella Figura 3.2, e fissiamo $MinPts = 4$. Il cerchio interno ha raggio ε, mentre quello esterno ha raggio $\varepsilon(1+\rho)$. I core point e i non-core point sono rispettivamente in bianco e nero. Il punto o_5 è ρ-approximate density-reachable da o_3 (tramite la sequenza o_3, o_2, o_1, o_5). Tuttavia, o_5 non è density-reachable da o_3.

Definizione 3.3. Un **ρ-approximate cluster** C è un sottoinsieme non vuoto di P tale che

- (Maximality) Se un core point $p \in C$, allora tutti i punti density-reachable da p appartengono anch'essi a C.
- (ρ-Approximate Connectivity) Per ogni coppia di punti $p_1, p_2 \in C$, esiste un punto $p \in C$ tale che sia p_1 che p_2 sono ρ-approximate density-reachable da p.

Si noti la differenza tra quanto scritto sopra e la definizione originale di cluster (Definizione 2.4): il requisito di connettività è stato indebolito nella ρ-approximate connectivity. Nella

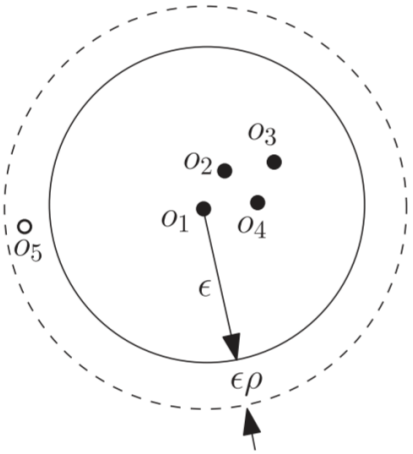


Figura 3.2: Density-reachability e ρ-approximate density-reachability ($MinPts = 4$).

Figura 3.2, entrambi gli insiemi $\{o_1, o_2, o_3, o_4\}$ e $\{o_1, o_2, o_3, o_4, o_5\}$ sono ρ-approximate cluster.

Problema 4. *Il **problema DBSCAN ρ-approximate** consiste nel trovare un insieme $\mathcal{C}$ di ρ-approximate cluster di P tali che ogni core point di P appartenga ad uno e un solo ρ-approximate cluster.*

A differenza del problema DBSCAN originale, la versione ρ-approximate potrebbe non avere un risultato univoco. Ad esempio, dato l'input della Figura 3.2, è consentito restituire sia $\{o_1, o_2, o_3, o_4\}$ che $\{o_1, o_2, o_3, o_4, o_5\}$. Tuttavia, qualsiasi risultato del problema ρ-approximate viene fornito con la garanzia di qualità che dimostreremo nella sezione successiva.

3.2 Sandwich Theorem

Sia DBSCAN che ρ-approximate DBSCAN sono parametrizzati da ε e $MinPts$. L'ideale sarebbe se potessero restituire esattamente gli stessi risultati di clustering. Naturalmente, questo non è possibile, tuttavia, mostreremo che è *quasi* vero: il risultato del ρ-approximate DBSCAN è garantito essere "compreso" tra i due risultati del DBSCAN esatto ottenuti dai parametri $(\varepsilon, MinPts)$ e $(\varepsilon(1+\rho), MinPts)$, in un senso che descriveremo di seguito. I cluster di DBSCAN raramente differiscono considerevolmente quando ε cambia soltanto di un piccolo fattore, infatti, se ciò accade, significa che la scelta di ε non è robusta, tanto che i cluster esatti non sono stabili.

Definiamo:

- $\mathcal{C}_1$ come l'insieme dei cluster di DBSCAN con parametri $(\varepsilon, MinPts)$
- $\mathcal{C}_2$ come l'insieme dei cluster di DBSCAN con parametri $(\varepsilon(1+\rho), MinPts)$
- $\mathcal{C}$ come un insieme arbitrario di cluster che è un risultato ammesso di $(\varepsilon, MinPts, \rho)$-approx-DBSCAN.

Il seguente teorema formalizza la garanzia di qualità menzionata in precedenza:

Teorema 3.4 (Garanzia di qualità a sandwich)**.** *Le seguenti affermazioni sono vere:*

(1) Per ogni cluster $C_1 \in \mathcal{C}_1$, esiste un cluster $C \in \mathcal{C}$ tale che $C_1 \subseteq C$.

(2) Per ogni cluster $C \in \mathcal{C}$, esiste un cluster $C_2 \in \mathcal{C}_2$ tale che $C \subseteq C_2$.

Dimostrazione. Per dimostrare l'Affermazione 1, sia p un core point arbitrario in C_1. Allora, C_1 è precisamente l'insieme di punti in P density-reachable da p. [1] In generale, se un punto q è density-reachable da p in $(\varepsilon, MinPts)$-exact-DBSCAN, q è anche density-reachable da p in $(\varepsilon, MinPts, \rho)$-approx-DBSCAN. Per la maximality della Definizione 3.3, se C è il cluster in $\mathcal{C}$ contenente p, allora tutti i punti di C_1 devono essere in C.
Per dimostrare l'Affermazione 2, sia p un core point arbitrario in C (deve essercene uno per la Definizione 3.3). Anche in $(\varepsilon(1+\rho), MinPts)$-exact-DBSCAN, p deve essere un core point. Scegliamo C_2 come cluster di $\mathcal{C}_2$ a cui appartiene p. Ora, fissiamo un punto arbitrario $q \in C$. In $(\varepsilon, MinPts, \rho)$-approx-DBSCAN, per ρ-approximate connectivity della Definizione 3.3, sappiamo che p e q sono entrambi ρ-approximate reachable da un punto z. Ciò implica che z è anche ρ-approximate reachable da p. Quindi, q è ρ-approximate reachable da p. Ciò significa che q è density-reachable da p in $(\varepsilon(1+\rho), MinPts)$-exact-DBSCAN, indicando che $q \in C_2$. □

Vediamo un'interpretazione alternativa e maggiormente intuitiva del Teorema 3.4:

[1] Dimostrazione. Per la maximality della Definizione 2.4, tutti i punti density-reachable da p sono in C_1. D'altra parte, sia q un qualsiasi punto di C_1. Per connectivity, p e q sono entrambi density-reachable da un punto z. Essendo che p è un core point, sappiamo che z è anche density-reachable da p. Quindi, q è density-reachable da p.

- L'Affermazione 1 asserisce che se due punti appartengono allo stesso cluster nel DBSCAN con parametri $(\varepsilon, MinPts)$, allora saranno sicuramente nello stesso cluster nel ρ-approximate DBSCAN con gli stessi parametri.

- D'altra parte, un cluster del ρ-approximate DBSCAN parametrizzato da $(\varepsilon, MinPts)$ può anche contenere due punti p_1, p_2 che appartengono a due cluster diversi nel DBSCAN con gli stessi parametri. A questo punto, l'Affermazione 2 asserisce che non appena il parametro ε aumenta a $\varepsilon(1+\rho)$, p_1 e p_2 saranno parte dello stesso cluster di DBSCAN.

In questo paragrafo discuteremo gli effetti dell'approssimazione. Quanti cluster si possono individuare nel dataset rappresentato nella Figura 3.3? Dipende dal valore di ε considerato, il quale ci permette di visualizzare il dataset da varie granularità, ottenendo diversi risultati di clustering. Nella Figura 3.3, dato ε_1 (e $MinPts = 2$), DBSCAN restituisce tre cluster. Dato ε_2, invece, DBSCAN restituisce due cluster, dovuto al fatto che, a questa distanza, i due cluster sulla destra si fondono in uno solo. Consideriamo ora l'approssimazione. Le circonferenze tratteggiate nella Figura 3.3 corrispondono ai raggi ottenuti con l'approssimazione ρ. Sia per ε_1 che per ε_2, DBSCAN ρ-approximate restituirà esattamente gli stessi cluster, poiché tali valori di distanza sono stati scelti con robustezza, dimostrandosi insensibili a perturbazioni di piccola entità. Per ε_3, tuttavia, DBSCAN ρ-approximate può restituire solo un cluster (ovvero tutti i punti del dataset appartengono allo stesso cluster), mentre il DBSCAN esatto ne restituirà due (gli stessi cluster di ε_2). Osservando la figura attentamente, si può notare che ciò accade perché il cerchio tratteggiato di raggio $(1+\rho)\varepsilon_3$ passa per un punto – in tal caso il punto O – che cade al di fuori del cerchio pieno di raggio ε_3. Per questo la scelta del parametro ε_3 può essere considerata scadente, poiché rappresenta un valore prossimo alla distanza tra due cluster, anche una minima variazione di ε_3 potrebbe avere un impatto significativo sui risultati di clustering. Introduciamo un utile corollario del sandwich theorem:

Corollario 3.5. *Siano $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}$ come definiti nel Teorema 3.4. Se un cluster C compare sia in $\mathcal{C}_1$ che in $\mathcal{C}_2$, allora C sarà un cluster anche in $\mathcal{C}$.*

Dimostrazione. Supponiamo, al contrario, che $\mathcal{C}$ non contenga C. Per il Teorema 3.4:

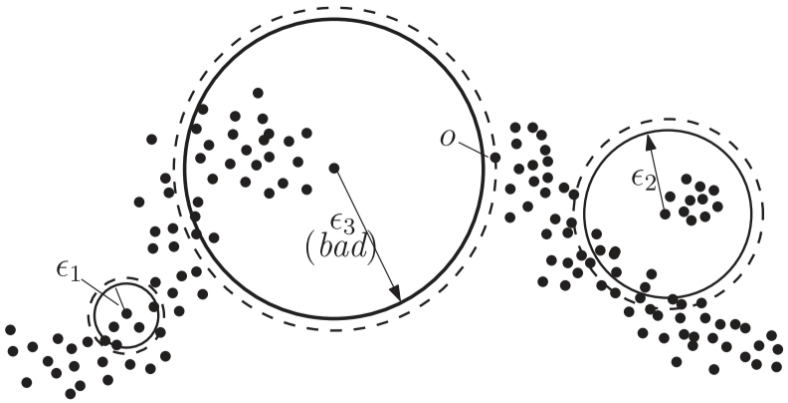


Figura 3.3: Buone e cattive scelte di ε.

(i) $\mathcal{C}$ deve contenere un cluster C' tale che $C \subseteq C'$.

(ii) $\mathcal{C}_2$ deve contenere un cluster C'' tale che $C' \subseteq C''$.

Questo implica che $C \subseteq C''$. Inoltre, essendo $C \in \mathcal{C}_2$, segue che, in $\mathcal{C}_2$, ogni core point in C appartiene anche a C''. Questo è impossibile perché un core point può appartenere a un solo cluster. □

Il corollario afferma che, sebbene alcuni dei cluster identificati tramite il DBSCAN esatto subiscano cambiamenti quando il valore di ε aumenta di un fattore $(1+\varepsilon)$ (ovvero, ε non è robusto), il DBSCAN ρ-approximated determina comunque tutti quei cluster che mantengono la loro struttura invariata. Per chiarire ulteriormente, consideriamo i punti presenti nella Figura 3.3 come parte di un dataset più ampio, dove i cluster associati agli altri punti rimangono immutati quando il parametro ε_3 cresce a $\varepsilon_3(1+\rho)$. Alla luce del Corollario 3.5, possiamo affermare con certezza che il ρ-approximate DBSCAN, applicato con il parametro ε_3, identificherà in modo accurato tutti questi cluster.

3.3 Calcolo approssimato del numero di punti

Consideriamo ora un altro problema, la cui risoluzione riveste un ruolo cruciale per il nostro algoritmo ρ-approximate DBSCAN.

Sia P un insieme di n punti in $\mathbb{R}^d$ dove d è una costante. Dato un qualsiasi punto $q \in \mathbb{R}^d$, una distanza di soglia $\varepsilon > 0$ e una costante arbitrariamente piccola $\rho > 0$, una *approximate range count query* restituisce un numero intero che è garantito essere compreso tra $|B(q,\varepsilon) \cap P|$ e $|B(q,\varepsilon(1+\rho)) \cap P|$. In altre parole, la query restituisce un valore intero

compreso tra il numero di punti interni alla palla di raggio ε e il numero di punti interni alla palla di raggio $\varepsilon(1+\rho)$, entrambe centrate nel medesimo punto q. Ad esempio, considerando la Figura 3.2, dato $q = o_1$, una query potrebbe restituire 4 o 5.
Arya e Mount [2] hanno sviluppato una struttura che occupa $O(n)$ spazio, la quale viene costruita in tempo $O(n \log n)$ e risponde a qualsiasi query della tipologia sopra discussa in tempo $O(\log n)$. Gan e Tao [6], invece, hanno progettato una struttura alternativa che risponde alle medesime query e che offre prestazioni migliori nel contesto specifico della nostra ricerca:

Lemma 3.6. *Per qualsiasi valori di ε e ρ fissati, esiste una struttura che occupa $O(n)$ spazio, la quale può essere costruita in un tempo atteso $O(n)$ e risponde a qualsiasi approximate range count query in un tempo atteso $O(1)$.*

In questa sezione, procederemo all'analisi di quest'ultima struttura.

Struttura. La struttura è un partizionamento di $\mathbb{R}^d$ a griglia gerarchica, simile ad un quadtree. Innanzitutto, consideriamo una griglia su $\mathbb{R}^d$ dove ogni cella è un ipercubo d-dimensionale con lato di lunghezza $\varepsilon/\sqrt{d}$. Qualsiasi cella c che contenga almeno un punto appartenente a P, viene suddivisa in 2^d celle della stessa dimensione, che avranno lato pari alla metà del lato della cella iniziale. Questo processo di suddivisione viene applicato in modo ricorsivo a ogni cella risultante che non sia vuota (ovvero, contiene almeno un punto di P), fino a quando la lunghezza del lato di ciascuna cella risulta essere al massimo $\varepsilon\rho/\sqrt{d}$.
Denominiamo con la lettera H la gerarchia così ottenuta. All'interno di H, manteniamo solo le celle non vuote, e per ognuna di queste celle, memorizziamo $cnt(c)$, un valore intero che rappresenta il numero di punti appartenenti a P contenuti all'interno della cella c. Definiamo una *cella di livello* i in H come una cella con un lato di lunghezza $\varepsilon/(2^i\sqrt{d})$. È importante notare che la gerarchia H ha un numero limitato di livelli, precisamente $h = \max\{1, 1 + \lceil \log_2(1/\rho) \rceil\}$, che risulta essere $O(1)$. Nel caso in cui una cella c' di livello $(i+1)$ sia contenuta all'interno di una cella c di livello i, si dice che c' è *figlia* di c, mentre c è il *genitore* di c'. Qualsiasi cella che non abbia figli viene chiamata *cella foglia*. La Figura 3.4 mostra una rappresentazione grafica dei primi tre livelli di H relativi al dataset sulla sinistra. È rilevante notare che le celle vuote non vengono memorizzate nella

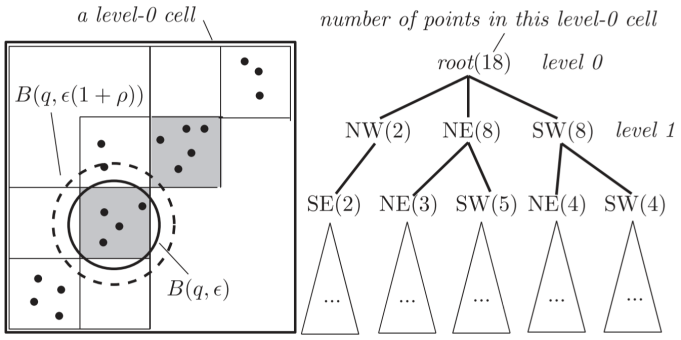


Figura 3.4: Calcolo approssimato del numero di punti compreso tra $|B(q,\varepsilon) \cap P|$ e $|B(q,\varepsilon(1+\rho)) \cap P|$.

struttura.

Query. Un'approximate range count query riceve in input i parametri q, ε, ρ e restituisce la risposta ans calcolata come segue. Inizialmente, si pone $ans = 0$. Successivamente, l'algoritmo considera le celle non vuote di livello 0 che intersecano la palla $B(q,\varepsilon)$ ed esegue la seguente procedura: data una cella c di livello i non vuota, si distinguono tre casi:

1. Se la cella c non interseca la palla $B(q,\varepsilon)$, tale cella si ignora.

2. Se la cella c è contenuta completamente dalla palla $B(q,\varepsilon(1+\rho))$, si aggiunge il valore $cnt(c)$ ad ans.

3. Quando nessuna delle condizioni precedenti è soddisfatta, si verifica se c è una cella foglia in H. In caso affermativo, si aggiunge $cnt(c)$ ad ans solo se la cella c interseca la palla $B(q,\varepsilon)$. Altrimenti (c non è una foglia), si processano le celle figlie di c con la medesima procedura.

Correttezza. L'algoritmo sopra descritto offre due garanzie fondamentali:

1. Se un punto $p \in P$ è interno alla palla $B(q,\varepsilon)$, è sicuramente contato in ans;

2. Se, invece, il punto p è esterno alla palla $B(q,\varepsilon(1+\rho))$, è garantito che non sarà contato in ans.

Dimostrazione. Siano q, ε, ρ i parametri di un'approximate range count query arbitraria. Verifichiamo le due garanzie sopra descritte:

1. Consideriamo un punto $p \in P$ interno alla palla $B(q, \varepsilon)$ e dimostriamo che sarà sicuramente contato in ans. La cella di livello 0 a cui appartiene il punto p verrà considerata dall'algoritmo in quanto interseca la palla $B(q, \varepsilon)$. A tal punto si potranno verificare due casi: il punto p è contenuto in una cella di livello i che è coperta completamente dalla palla $B(q, \varepsilon(1+\rho))$ e quindi il punto sarà contato in ans oppure il punto p è contenuto in una cella foglia ed essendo che quest'ultima interseca la palla $B(q, \varepsilon)$, il punto sarà contato in ans.

2. Consideriamo ora un punto $p \in P$ esterno alla palla $B(q, \varepsilon(1+\rho))$ e dimostriamo che non sarà contato in ans. Se la cella di livello 0 a cui appartiene il punto p non interseca la palla $B(q, \varepsilon)$, questa non sarà considerata dall'algoritmo e di conseguenza il punto p non sarà contato in ans. In caso contrario, si potranno verificare due casi: il punto p è contenuto in una cella di livello i che non interseca la palla $B(q, \varepsilon)$, tale cella verrà ignorata dall'algoritmo e quindi il punto non sarà contato in ans, oppure il punto p è contenuto in una cella foglia, ma essendo che il punto è esterno alla palla $B(q, \varepsilon(1+\rho))$, tale cella non potrà mai intersecare la palla $B(q, \varepsilon)$ perché due punti qualsiasi in una cella foglia sono entro la distanza $\varepsilon\rho$. Anche in questo caso, il punto non verrà contato in ans.

Ne consegue che il valore di ans restituito è una risposta valida. □

Complessità temporale. Si tenga presente che la gerarchia H è caratterizzata da un numero costante di livelli, che possiamo considerare $O(1)$. Dato che ogni livello ha un numero $O(n)$ di celle non vuote, lo spazio occupato dalla struttura H è $O(n)$. Grazie alla tecnica dell'hashing, è possibile costruire la struttura, livello per livello, in un tempo previsto $O(n)$.
Per l'analisi della complessità temporale dell'algoritmo di query, è opportuno notare che ogni cella c visitata dalla procedura deve soddisfare una delle seguenti condizioni:

1. c è una cella di livello 0;

2. il genitore di c interseca il bordo di $B(q, \varepsilon)$.

Nel primo caso, è possibile individuare le $O(1)$ celle di livello 0 che intersecano $B(q, \varepsilon)$ in un tempo previsto $O(1)$, utilizzando le coordinate del punto q. Nel secondo caso, è suf-

ficiente trovare un limite superiore al numero di celle che intersecano il confine di $B(q,\varepsilon)$, in quanto ciascuna di queste celle ha un numero costante di nodi figli, pari a $2^d = O(1)$. In generale, considerando una griglia d-dimensionale composta da celle di lato ℓ, è noto, grazie al lavoro di Arya e Mount [2], che il numero di celle che intersecano il bordo di una palla avente raggio θ è pari a

$$O\left(1+\left(\frac{\theta}{\ell}\right)^{d-1}\right)$$

Unendo questo risultato con il fatto che una cella di livello i ha lati di lunghezza $\varepsilon/(2^i\sqrt{d})$, possiamo dedurre che il numero totale di celle (considerando tutti i livelli) che intersecano la frontiera di una palla $B(q,\varepsilon)$ è limitato superiormente da:

$$\begin{aligned}\sum_{i=0}^{h-1} O\left(1+\left(\frac{\varepsilon}{\varepsilon/\left(2^i\sqrt{d}\right)}\right)^{d-1}\right) &= \sum_{i=0}^{h-1} O\left(\left(2^i\right)^{d-1}\right)\\ &= O\left(\left(2^h\right)^{d-1}\right)\\ &= O\left(1+(1/\rho)^{d-1}\right)\end{aligned}$$

dove l'ultimo passaggio è ottenuto sostituendo il numero di livelli della struttura H uguale a $h = \max\{1, 1+\left\lceil \log_2(1/\rho)\right\rceil\}$. La complessità temporale ottenuta è costante per ogni ρ fissato.

3.4 Risoluzione del problema ρ-approximate DBSCAN

A questo punto, siamo in possesso di tutti gli strumenti necessari per affrontare la risoluzione del problema ρ-approximate DBSCAN e procedere con la dimostrazione del seguente teorema.

Teorema 3.7. *Esiste un algoritmo ρ-approximate DBSCAN che termina in tempo $O(n)$, indipendentemente dal valore di ε, dal coefficiente di approssimazione costante ρ e dalla dimensione fissata d.*

3.4.1 Algoritmo

Inizialmente, si costruisce una griglia T in $\mathbb{R}^d$ dove ogni cella è un ipercubo d-dimensionale con lato $\varepsilon/\sqrt{d}$. Questo assicura che due punti appartenenti alla stessa cella abbiano una

distanza minore o uguale ad ε. Una cella c della griglia T è *non-vuota* se contiene almeno un punto dell'insieme P; altrimenti, c è vuota. È importante notare che ci possono essere al massimo n celle non-vuote.
Successivamente l'algoritmo esegue una procedura per ogni punto $p \in P$ al fine di determinare se p è un core point o viceversa un non-core point.
Indichiamo con $P(c)$ l'insieme dei punti appartenenti all'insieme P e contenuti nella cella c. Una cella c è chiamata *core cell* se $P(c)$ contiene almeno un core point. Definiamo S_{core} come l'insieme delle core cell nella griglia T.
Definiamo ora un grafo $G = (V, E)$ come segue:

- Ogni vertice in V corrisponde ad una core cell distinta della griglia T.
- Date due diverse core cell $c_1, c_2 \in S_{core}$, l'esistenza di un arco nell'insieme E tra le celle c_1 e c_2 è determinata dalle seguenti regole:
  - *Affermativo.* L'arco tra c_1 e c_2 è presente se esistono almeno due core point p_1, p_2 appartenenti, rispettivamente, alle celle c_1, c_2, tali che $dist(p_1, p_2) \leq \varepsilon$;
  - *Negativo.* L'arco tra c_1 e c_2 è assente se nessun core point in c_1 si trova entro la distanza $\varepsilon(1+\rho)$ da un qualsiasi core point in c_2;
  - *Indifferente.* In tutti gli altri casi, la presenza o l'assenza dell'arco tra c_1 e c_2 rimane indefinita.

A tal punto, l'algoritmo individua tutte le componenti connesse del grafo G. Indichiamo con k il numero delle componenti connesse, con $V_i (1 \leq i \leq k)$ l'insieme dei vertici appartenenti alla i-esima componente connessa e con $P(V_i)$ l'insieme dei core point appartenenti alle celle dell'insieme V_i. Allora:

Lemma 3.8. *Il numero k delle componenti connesse corrisponde al numero di ρ-approximate cluster individuati dall'algoritmo. Inoltre, $P(V_i)(1 \leq i \leq k)$ è esattamente l'insieme dei core point appartenenti all'i-esimo cluster.*

L'algoritmo ha quindi individuato per ogni core point il cluster a cui appartiene, come ultimo passaggio, per ogni non-core point, si controlla se può essere assegnato ai cluster esistenti o meno. Eseguita questa procedura, l'algoritmo ha individuato i cluster dell'insieme P in maniera approssimata e termina.

Analizziamo ora, nel dettaglio, come l'algoritmo esegue le procedure sopra citate.

Costruzione della griglia. La griglia T viene creata in modo dinamico: per ogni punto appartenente all'insieme P si determina la cella di appartenenza dividendo le coordinate per la lunghezza del lato delle celle. Si esegue un controllo per determinare se la cella è già esistente, in caso contrario viene creata. Infine, si aggiunge il punto ad essa.

Determinare i core point. Siano c_1 e c_2 due celle distinte nella griglia T. Sono considerate ε-vicine se la minima distanza tra loro è minore o uguale ad ε. Se una cella non vuota c contiene un numero di punti uguale o maggiore al valore del parametro $MinPts$, allora tutti questi punti sono core point.

Consideriamo il caso rimanente, ovvero una cella c dove $|P(c)| < MinPts$. Ciascun punto $p \in P(c)$ può essere o meno un core point. Per determinarlo, l'algoritmo segue una procedura basata sulla definizione: per ogni punto $p \in P(c)$, controlla se il numero di punti contenuti nella palla $B(p, \varepsilon)$ è maggiore o uguale a $MinPts$, in caso affermativo p sarà un core point. Chiaramente si ha già un numero di punti uguale a $|P(c)|$, che sono contenuti nella palla $B(p, \varepsilon)$. Per trovare i restanti punti, è sufficiente calcolare le distanze tra il punto p e i punti contenuti nelle celle ε-vicine alla cella c. Da notare che una volta trovati un numero di punti contenuti nella palla $B(p, \varepsilon)$ pari a $MinPts$, si può terminare la ricerca e classificare il punto p come un core point.

Creazione del grafo G. Per ottenere il grafo G, l'algoritmo inizia costruendo, per ogni core cell c appartenente alla griglia T, la struttura menzionata nel Lemma 3.6. Questa struttura viene costruita considerando l'insieme dei core point contenuti nella cella c.

Per generare gli archi di una core cell c_1, si processano iterativamente tutte le celle c_2 che sono ε-vicine alla cella c_1. Per ogni core point p contenuto nella cella c_1, si esegue una approximate range count query sull'insieme dei core point contenuti nella cella c_2. Se la query restituisce un risultato diverso da zero, si aggiunge un arco (c_1, c_2) nel grafo G.

Qualora avessimo processato tutti i core point appartenenti a c_1 senza aver aggiunto alcun arco, giungiamo alla conclusione che non esisterà alcun collegamento tra la cella c_1 e la cella c_2.

Determinare i cluster. I cluster vengono determinati trovando le componenti connesse del grafo G. È quindi sufficiente utilizzare un algoritmo per trovare le componenti connesse in un grafo.

Clustering dei border point. È importante sottolineare che ciascun insieme $P(V_i)(1 \leq i \leq k)$ contiene solamente i core point appartenenti all'i-esimo cluster di P. Inoltre, si ricorda che un non-core point può risultare associato a nessun cluster, in tal caso viene definito un *noise point*, oppure può appartenere a uno o più cluster, assumendo la denominazione di *border point*. È quindi necessario assegnare ogni non-core point q (ovvero un border point) al cluster appropriato. Tale assegnazione si basa sul seguente principio: *se p è un core point e $dist(p,q) \leq \varepsilon$, allora q deve essere aggiunto al cluster (univoco) a cui appartiene p.* Per individuare tutti questi core point p, Gunawan [8] ha adottato il seguente algoritmo: sia c la cella a cui appartiene il punto q, per ogni cella c' che è ε-vicina alla cella c, si calcola la distanza tra il punto q e tutti i core point contenuti nella cella c'.

3.4.2 Correttezza

Consideriamo un cluster arbitrario C restituito dall'algoritmo sopra descritto. Dimostriamo che il cluster C soddisfa la Definizione 3.3.

Maximality. Sia p un core point arbitrario in C, e q un punto qualsiasi appartenente a P, tale che sia density-reachable dal punto p. Dimostreremo che q appartiene al cluster C. Consideriamo due casi:

- Il punto q è un core point. Per la Definizione 2.3, esiste una sequenza di core point $p_1, p_2, \ldots, p_t$ (per qualche intero $t \geq 2$) tale che $p_1 = p$, $p_t = q$ e $dist(p_i, p_{i+1}) \leq \varepsilon$ per ogni $i \in [1, t-1]$. Indichiamo con c_i la cella della griglia T che contiene il punto p_i. Per come è definito il grafo G, ci deve essere un arco tra c_i e c_{i+1} per ogni $i \in [1, t-1]$. Di conseguenza c_1 e c_t devono appartenere alla stessa componente connessa nel grafo G; pertanto, p e q devono appartenere allo stesso cluster.

- Il punto q è un non-core point. La correttezza, in questo caso, è banalmente garantita dalla procedura con cui vengono assegnati i non-core point ai cluster.

ρ-Approximate Connectivity. Sia p un core point arbitrario in C. Per ogni punto $q \in C$, dimostreremo che q è ρ-approximate density-reachable da p. Consideriamo due casi:

- Il punto q è un core point. Indichiamo con c_p e c_q le celle della griglia T a cui appartengono rispettivamente i punti p e q. Poiché c_p e c_q appartengono alla stessa componente connessa di G, esiste un cammino $c_1, c_2, \ldots, c_t$ in G (per qualche

intero $t \geq 2$) tale che $c_1 = c_p$ e $c_t = c_q$. Ricordiamo che due punti qualsiasi appartenenti alla stessa cella sono distanti al più ε. Unendo questa considerazione con la definizione degli archi del grafo G, possiamo affermare che esiste una sequenza di core point $p_1, p_2, ..., p_{t'}$ (per qualche intero $t' \geq 2$) tale che $p_1 = p, p_{t'} = q$ e $dist(p_{i+1}, p_i) \leq \varepsilon(1+\rho)$ per ogni $i \in [1, t'-1]$. Possiamo quindi concludere che q è ρ-approximate density-reachable da p.

- Il punto q è un non-core point. Anche in questo caso, la correttezza è banalmente garantita dalla procedura con cui vengono assegnati i non-core point ai cluster.

3.4.3 Complessità temporale

Determiniamo la complessità temporale dell'algoritmo andando ad analizzare le diverse fasi che lo compongono:

Costruzione della griglia. Si esegue un'iterazione su tutti i punti dell'insieme P per determinare la cella di appartenenza, questa operazione ha una complessità temporale pari a $O(n)$. Le operazioni di memorizzazione e ricerca delle celle, utilizzando la tecnica dell'hashing, possono essere eseguite in tempo costante $O(1)$. La complessità temporale per la costruzione della griglia T è quindi $O(n)$.

Determinare i core point. Per ogni punto $p \in P$, l'algoritmo deve stabilire se è un core point o meno. Sia c la cella che contiene il punto p, nel caso $P(c) \geq MinPts$ si può affermare fin da subito che p è un core point, nel caso rimanente è necessario visitare le celle ε-vicine alla cella c. Indichiamo con C_1 l'insieme delle celle c tali che $P(c) \geq MinPts$ e con C_2 l'insieme delle celle c tali che $P(c) < MinPts$. Per ogni cella in C_1 si devono iterare i punti al suo interno e si impiega $O(P(c))$ tempo, quindi per processare tutte queste celle si ha una complessità temporale pari a $O(n)$. Per quanto riguarda ogni cella c in C_2, invece, dobbiamo iterare i punti contenuti nelle celle ε-vicine a c_2. Indicando con $\mathcal{V}_\varepsilon(c)$ l'insieme delle celle ε-vicine alla cella c, possiamo esprimere la complessità temporale per processare le celle in C_2 come:

$$\sum_{c \in C_2} \sum_{c' \in \mathcal{V}_\varepsilon(c)} O(P(c)P(c')) \leq O(minPts) \cdot \sum_{c \in C_2} \sum_{c' \in \mathcal{V}_\varepsilon(c)} O(P(c'))$$

Consideriamo una cella c', i punti al suo interno sono processati soltanto da celle c che hanno la cella c' come ε-vicina, ovvero $c' \in \mathcal{V}_\varepsilon(c)$, che è equivalente a $c \in \mathcal{V}_\varepsilon(c')$. Possiamo quindi scrivere la complessità come:

$$\begin{aligned} O(minPts) \cdot \sum_{c \in C_2} \sum_{c' \in \mathcal{V}_\varepsilon(c)} O(P(c')) &\leq O(minPts) \cdot \sum_{c' \in C_1 \cup C_2} \sum_{c \in \mathcal{V}_\varepsilon(c')} O(P(c')) \\ &\leq O(minPts) \cdot \sum_{c' \in C_1 \cup C_2} O(P(c')) \cdot |\mathcal{V}_\varepsilon(c)| \\ &\leq O(minPts) \cdot O(n) \cdot |\mathcal{V}_\varepsilon(c)| \\ &\leq O(minPts \cdot n) \end{aligned}$$

Nell'ultimo passaggio, giungiamo a questa conclusione considerando il numero di celle ε-vicine, vale a dire $|\mathcal{V}_\varepsilon(c)|$, una quantità costante $O(1)$, fissata la dimensionalità d. In conclusione, la complessità temporale per determinare i core point è $O(minPts \cdot n)$.

Creazione del grafo G. Per costruire la struttura descritta nel Lemma 3.6 si impiega un tempo $O(n)$ per tutte le celle. Il tempo necessario per la creazione del grafo G è direttamente proporzionale al numero di approximate range count query effettuate. Per ciascun core point contenuto in una cella c_1, eseguiamo una query per ogni cella ε-vicina alla cella c_1. Considerato che il numero di celle ε-vicine ad un'altra cella è costante, per ciascun core point vengono effettuate $O(1)$ query. Di conseguenza, il numero totale di query è $O(n)$.

Determinare i cluster. Per determinare i cluster si utilizza un algoritmo per trovare le componenti connesse in un grafo, come la breadth-first search, oppure si può ricorrere anche alla struttura dati disjoin-set. Entrambi gli approcci sono descritti in [3] e hanno complessità temporale lineare.

Clustering dei border point. Per ogni non-core point $q \in P$, sia c la cella a cui appartiene il punto q, si visitano tutte le celle ε-vicine alla cella c per calcolare la distanza tra il punto q e i core point contenuti in tali celle. Considerato che il numero di non-core point contenuti in una cella sarà minore di $MinPts$, si può ricondurre il calcolo della complessità temporale al procedimento eseguito sopra nel caso dell'assegnazione dei core point. Possiamo quindi concludere che la complessità totale per assegnare i border point ai cluster appropriati è $O(minPts \cdot n)$.

La complessità temporale dell'algoritmo ρ-approximate DBSCAN è quindi pari a $O(n) + O(minPts \cdot n) + O(n) + O(minPts \cdot n) = O(minPts \cdot n) = O(n)$.

Osservazione. È importante sottolineare che la complessità temporale di $O(n)$ ha una costante nascosta dell'ordine di $(1/\rho)^{d-1}$, dovuta alle approximate range count query e provata nella dimostrazione del Lemma 3.6. Considerato che tale costante è esponenziale nella dimensione d, l'algoritmo approssimato è efficiente solo per valori piccoli di d.

Capitolo 4

Studio di un'implementazione

In questo capitolo studieremo un'implementazione dell'algoritmo ρ-approximate DBSCAN, scritta nel linguaggio di programmazione C++ [7]. Tale implementazione è stata progettata da Gan e Tao in occasione della conferenza ACM SIGMOD dell'anno 2015. In tale circostanza, il loro contributo è stato insignito del prestigioso "Best Paper Award". Per ciascuna delle fasi dell'algoritmo presentate nel capitolo precedente, effettueremo un'analisi dettagliata delle strutture dati e degli algoritmi utilizzati.

4.1 Costruzione della griglia

Ricordiamo che la griglia T è costituita da celle di livello 0, che corrispondono a degli ipercubi d-dimensionali con lato di lunghezza $\varepsilon/\sqrt{d}$. Per memorizzare tali celle viene utilizzata una `hash_map`, definita nella libreria `ext/hash_map` di gcc e appartenente alla namespace `__gnu_cxx`. Il valore di hash di una cella viene calcolato andando a moltiplicare ognuna delle d coordinate per un numero intero casuale e sommando tutti questi valori. Affinché tale valore possa essere memorizzato in una variabile `unsigned int`, nello svolgimento dei calcoli si fa ricorso all'algebra modulare, utilizzando come modulo un numero primo sufficientemente grande. Per quanto riguarda la chiave dell'`hash_map`, viene utilizzato un vettore di interi che contiene le d coordinate spaziali della cella e come ultimo elemento, il valore di hash.

Come spiegato precedentemente, la griglia T viene creata in modo dinamico. Per ogni punto p appartenente all'insieme P, si dividono le sue d coordinate spaziali per la lunghezza del lato di una cella, pari a $\varepsilon/\sqrt{d}$ e si ottengono così le d coordinate che individuano la

cella di livello 0 che conterrà il punto p. Si calcola il valore di hash della cella e si controlla se è già presente nella `hash_map`, in caso negativo viene creata una nuova cella con le coordinate corrispondenti e inserita nella struttura dati. A tal punto, si può aggiungere il punto alla relativa cella.

4.1.1 Celle ε-vicine

Un'operazione di fondamentale importanza per il nostro algoritmo è l'accesso alle celle ε-vicine ad una data cella c. Tale operazione è utile nei seguenti scenari:

- determinare i core point, in particolare quando un punto p appartiene ad una cella che contiene un numero di punti inferiore a $MinPts$;
- determinare gli archi della cella c all'interno del grafo G;
- assegnare i non-core point ai cluster appropriati.

Al fine di eseguire questa operazione, le celle di livello 0, oltre ad essere memorizzate nella `hash_map`, sono inserite anche in un `set`, una struttura dati implementata nella libreria standard del linguaggio C++. Tale struttura è implementata tramite un albero binario di ricerca e, nel nostro caso, utilizziamo come ordinamento delle celle il z-order [10]. Si tratta di un ordinamento che permette di mappare i dati multidimensionali, come le coordinate delle celle, in dati monodimensionali, preservando la loro località spaziale. Il z-value di un punto multidimensionale viene semplicemente calcolato alternando i bit delle rappresentazioni binarie di ogni coordinata. Nel nostro caso, non ci interessa calcolare i z-value delle celle, piuttosto è necessario confrontare due celle e determinare quale viene prima in tale ordinamento. Per effettuare tale confronto, per ciascuna dimensione, si considera il bit più significativo del risultato dell'operazione di "OR esclusivo" (indicata come $\wedge$ in C++) tra le due coordinate delle celle relative alla dimensione in esame. Si considera poi la dimensione per cui la posizione del bit più significativo è maggiore e si confrontano soltanto le due coordinate di tale dimensione per determinare il z-order delle due celle. Riprendiamo ora l'operazione di nostro interesse ovvero trovare le celle ε-vicine ad una data cella c. Per ogni dimensione, sia q la coordinata della cella c in tale dimensione, si calcola il range di coordinate entro le quali risiedono le celle ε-vicine, che corrisponde all'intervallo $[q - \lceil\sqrt{d}\rceil, q + \lceil\sqrt{d}\rceil]$. Si ottengono quindi un vettore che rappresenta il limite

inferiore per le coordinate delle celle ϵ-vicine e uno che rappresenta il limite superiore, a tal punto si può effettuare la query nell'albero binario di ricerca. Tuttavia, effettuando una ricerca su dei dati multidimensionali, la ricerca binaria non è efficiente. Affinché la ricerca all'interno di un intervallo sia efficiente è necessario un algoritmo per calcolare, a partire dalle coordinate correnti, il prossimo z-value che si trova nell'intervallo di ricerca, chiamato BIGMIN. Invece, sempre a partire dalle coordinate correnti, il precedente z-value appartenente all'intervallo di ricerca, prende il nome di LITMAX. Tale algoritmo è stato progettato da Tropf e Herzog [10]. Riassumendo, data una cella c, per trovare le sue celle ε-vicine, si devono calcolare i due vettori che rappresentano il limite inferiore e superiore per le coordinate delle celle ε-vicine: tali vettori individuano il nostro campo di ricerca. Successivamente si applica l'algoritmo sopra citato all'albero binario di ricerca e si ottengono le celle desiderate. Da notare che il campo di ricerca contiene anche celle aggiuntive, quindi una volta trovata una cella appartenente ad esso, bisogna verificare che sia ε-vicina alla cella c.

4.2 Determinare i core point

Consideriamo lo pseudocodice sottostante, che descrive l'algoritmo per determinare i core point. La funzione riceve in ingresso la struttura all'interno della quale sono memorizzate le celle di livello 0 e i due parametri ε e $MinPts$. Successivamente, si segue la procedura descritta nel capitolo precedente, ovvero, per ogni cella di livello 0, nel caso questa contenga un numero di punti maggiore o uguale al parametro $MinPts$, allora tutti questi punti saranno core point. Altrimenti è necessario, per ogni punto pt contenuto nella cella corrente, andare a visitare le celle ε-vicine della cella in esame e contare quanti punti sono contenuti nella palla di raggio ε centrata nel punto pt. Una piccola ottimizzazione presente nell'algoritmo a riga 11, consiste nell'evitare di processare la cella c a cui appartiene il punto pt, essendo che i punti contenuti in essa sono sicuramente contenuti nella palla $B(pt, \varepsilon)$. Alla riga 20, si controlla se il numero di punti interni alla palla trovati finora è maggiore o uguale a $MinPts$, in caso affermativo si può interrompere la ricerca. Infine, alla riga 22, si esegue il controllo finale per determinare se il punto pt è un core point o meno.

Algoritmo 1: Determinare i core point

```
1  Funzione determinaCorePts(M, ε, minPts)
2      { M è la mappa in cui sono salvate le celle di livello 0 }
3      for ogni cella c ∈ M do
4          if P(c) ≥ minPts then
5              Contrassegna tutti i punti p ∈ c come core point
6          else
7              ℓ ← trovaCelleEpsVicine(c)
8              for ogni punto pt ∈ c do
9                  nPts ← P(c)
10                 for ogni cella c' ∈ ℓ do
11                     if c' ≠ c then
12                         { funzione che verifica l'intersezione tra la cella c'
                           e la palla di raggio ε centrata nel punto pt }
                           stato ← intersezioneConPalla(c', pt, ε)
13                         if state = 1 then
14                             { la cella c' è contenuta completamente dalla palla }
15                             nPts ← nPts + P(c')
16                         else
17                             for ogni punto y ∈ c' do
18                                 if dist(y, pt) ≤ ε then
19                                     nPts ← nPts + 1
20                         if nPts ≥ MinPts then
21                             break
22                 if nPts ≥ MinPts then
23                     Contrassegna pt come core point
```

4.3 Creazione del grafo G e identificazione dei cluster

Prima di eseguire la procedura per la creazione del grafo, per ogni cella che contiene almeno un core point, si costruisce la struttura, descritta nel Lemma 3.6, per il calcolo approssimato del numero di punti contenuti in una palla di raggio ε. Dopo questo passaggio, viene invocato l'algoritmo per costruire il grafo G, che esegue la procedura definita nel capitolo precedente. Per ragioni di efficienza, il grafo viene costruito implicitamente, ovvero non viene salvato in una lista di adiacenza, bensì viene costruito in modo progressivo e contemporaneamente si effettua l'identificazione dei cluster. Tale algoritmo è descritto dallo pseudocodice seguente:

Algoritmo 2: Creazione del grafo G

```
1  Funzione costruisciGrafo(M,ε)
2      { M è la mappa in cui sono salvate le celle di livello 0 }
3      Q ← ∅
4      clusterID ← 1
5      for ogni cella c ∈ M do
6          if NOT c.processed AND c.numCorePts > 0 then
7              Q.insert(c)
8              c.processed ← true
9              while Q non è vuota do
10                 cur ← Q.front()
11                 Q.pop()
12                 {funzione che assegna tutti i core point in cur al cluster corrente}
13                 assegnaCorePts(cur, clusterID)
14                 ℓ ← trovaCelleEpsVicine(cur)
15                 for ogni cella c' ∈ ℓ do
16                     if c'.numCorePts > 0 then
17                         arco ← verificaArco(cur, c', ε)
18                         if arco = true then
19                             if NOT c'.processed then
20                                 Q.insert(c')
21                                 c'.processed ← true
22             clusterID ← clusterID + 1
```

La funzione prende in ingresso l'`hash_map` dove sono salvate le celle di livello 0 e il parametro ε. Inizialmente, alla riga 3, viene dichiarata una coda con politica FIFO (First In First Out) e nella riga successiva viene dichiarata la variabile *clusterID*, che rappresenta l'identificativo del cluster corrente e che verrà progressivamente incrementata alla scoper-

ta di un nuovo cluster. Successivamente, si iterano le celle di livello 0 e se la cella corrente non è stata ancora processata e contiene almeno un core point, viene inserita nella coda. Trovata la prima cella che rispetta tali requisiti, alla riga 9 inizia la creazione del grafo implicito. Tramite un ciclo while, che itera finché la coda non è vuota, si estrae la cella cur che si trova in testa alla coda. Viene invocata la funzione *assegnaCorePts* che attribuisce a tutti i core point contenuti nella cella cur, l'appartenenza al cluster corrente, identificato dalla variabile *clusterID*. A seguire, alla riga 15, per ogni cella c' ε-vicina alla cella cur, se c' è una core cell, tramite la funzione *verificaArco* si va a controllare se le due celle cur e c' sono collegate da un arco. Tale funzione, facendo ricorso ad un'*approximate range count query*, va a verificare se le due celle soddisfano i requisti descritti nel capitolo precedente. In caso affermativo, non viene aggiunto nessun arco, bensì è sufficiente inserire la cella c' all'interno della coda e passare alla successiva iterazione del ciclo while. Quando termina il ciclo while, ovvero una volta svuotata la coda, significa che è terminata l'esplorazione del cluster corrente e si può passare al successivo, incrementando l'identificativo. Va evidenziato che, a discapito della generazione implicita del grafo, ciascun arco viene sottoposto a verifica in due occasioni, precisamente per ognuna delle due celle che esso connette.

4.4 Clustering dei border point

Infine, andiamo ad analizzare l'algoritmo che si occupa di processare i non-core point ed assegnarli agli eventuali cluster. Si ricorda che un non-core point può non rientrare in nessun cluster, come essere assegnato ad uno o più cluster. L'algoritmo è descritto dallo pseudocodice sottostante.

La funzione ha come parametri di input la mappa M dove sono salvate le celle di livello 0 e i parametri ε e $MinPts$. L'algoritmo inizia iterando le celle di livello 0 e controlla se la cella corrente c contiene almeno un non-core point. In caso affermativo, invoca la funzione *trovaCelleEpsVicine* per ottenere le celle ε-vicine alla cella c. Successivamente, per ogni non-core point p contenuto nella cella c, si vanno a scorrere le celle ε-vicine e per ognuna di esse, se contiene almeno un core point, si va a controllare, tramite la funzione *assegnabile*, se il punto p può essere assegnato allo stesso cluster dei core point contenuti in tale cella. Infine, dopo aver iterato tutte le celle ε-vicine, si controlla se la cella c alla

Algoritmo 3: Clustering dei border point

```
1  Funzione assegnaNonCorePts(M, ε, MinPts)
2      for ogni cella c ∈ M do
3          { se la cella c contiene almeno un non-core point }
4          if c.numCorePts < c.numPts then
5              ℓ ← trovaCelleEpsVicine(c)
6              for ogni non-core point p ∈ c do
7                  for ogni cella c' ∈ ℓ do
8                      if c ≠ c' AND c'.numCorePts > 0 then
9                          if assegnabile(p, c', ε) then
10                             ⋆ assegna il punto p allo stesso cluster
                               dei core point contenuti in c' ⋆
11             if c.numCorePts > 0 then
12                 ⋆ assegna il punto p allo stesso cluster
                   dei core point contenuti in c ⋆
```

quale appartiene il punto p contiene almeno un core point, in caso affermativo si assegna il punto p al cluster dei core point contenuti in tale cella. In questo frangente non è necessario alcun controllo, in quanto un non-core point è sicuramente density-reachable da un core point contenuto nella medesima cella, quindi il non-core point deve essere assegnato al cluster a cui appartiene il core point.

Capitolo 5

Analisi su alcuni dataset

In questo ultimo capitolo vengono analizzate le prestazioni relative all'implementazione dell'algoritmo ρ-approximate DBSCAN, citata nell'introduzione del capitolo precedente. Nella prima sezione andremo ad esaminare la qualità dei cluster generati dall'algoritmo approssimato a confronto con quelli individuati dall'algoritmo esatto. Nella seconda sezione si andranno ad analizzare i tempi di esecuzione su diversi dataset e al variare dei parametri. Sia l'implementazione dell'algoritmo esatto che quella dell'algoritmo approssimato sono fornite da Gan e Tao [7]. Tutti gli esperimenti sono stati eseguiti su un computer con le seguenti specifiche:

- Processore: Intel Core i7 CPU @2.80GHz
- RAM 16 GB
- Sistema operativo: Windows 11 Home 64-bit

5.1 Qualità dell'algoritmo approssimato

In questa sezione, andiamo ad osservare i risultati di clustering dell'algoritmo approssimato e li confrontiamo con quelli ottenuti dall'algoritmo esatto. Affinché sia possibile visualizzare graficamente i risultati, per questi test, viene utilizzato un dataset a due dimensioni ($d = 2$) contenente $n = 1000$ punti. Si tratta di un dataset sintetico, generato dall'algoritmo *Seed Spreader* fornito da Gan e Tao [7]. Nella Figura 5.1 sono riportati, in forma grafica, i cluster ottenuti, ognuno identificato da un colore differente. Per ottenere tali risultati è stato utilizzato lo stesso dataset ed è stato fissato il parametro $MinPts = 10$. Ad ogni riga

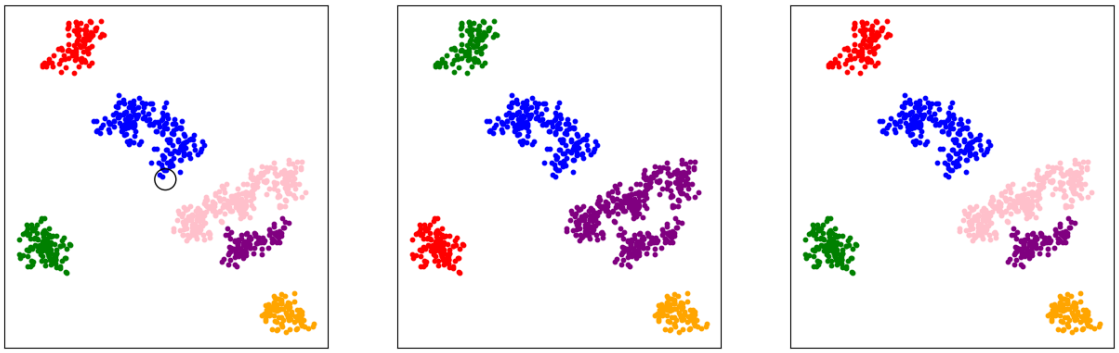
di immagini corrisponde un valore di ε fissato, rispettivamente $\varepsilon = 50, 100, 150$. La prima immagine di ogni riga, la quale contiene anche una circonferenza di raggio ε, rappresenta i risultati di clustering ottenuti dall'algoritmo esatto, mentre le restanti due immagini corrispondono all'algoritmo approssimato. Nella prima riga, con $\varepsilon = 50$, si può notare come l'algoritmo esatto individua 6 cluster diversi, invece quello approssimato con $\rho = 1$, fonde due cluster vicini e ne riconosce soltanto 5; è necessario diminuire il parametro ρ a 0.1 per trovare i 6 cluster.

Nella seconda riga, dove $\varepsilon = 100$, l'algoritmo esatto determina 5 cluster, il ρ-approximate DBSCAN con $\rho = 1$ fonde i tre cluster in alto e ritorna soltanto 3 cluster, anche in questo caso, è sufficiente diminuire il parametro ρ a 0.1 per ottenere i 5 cluster. Infine nella terza riga, con $\varepsilon = 150$, la versione esatta individua soltanto 2 cluster, in questo caso l'algoritmo approssimato è in grado di riconoscere i medesimi 2 cluster con il parametro ρ pari a 1 e logicamente anche con $\rho = 0.1$.

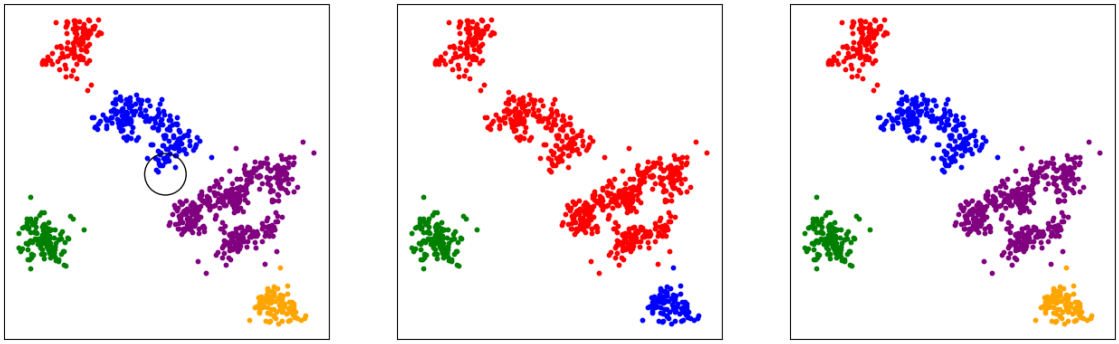
È interessante osservare anche la differenza dei risultati di clustering al variare del parametro ε. La scelta del parametro $\varepsilon = 50$ consente di analizzare il dataset nel dettaglio e ottenere un'accurata divisione in cluster. Aumentando ε ad un valore pari a 100, si diminuisce il livello di dettaglio e si vanno a unire due cluster, piuttosto vicini tra loro, che nel caso precedente erano distinti. Infine, con $\varepsilon = 150$, si ottiene un analisi piuttosto macroscopica, dove 4 cluster che precedentemente erano distaccati vengono uniti in uno solo. Quest'ultima scelta del parametro ε si può considerare scadente, restituendo un'analisi poco significativa. Si noti, inoltre, che tale valore è prossimo a $\varepsilon = 190$, per il quale l'algoritmo esatto racchiude tutti i punti del dataset in un unico cluster.

5.2 Analisi del tempo di esecuzione

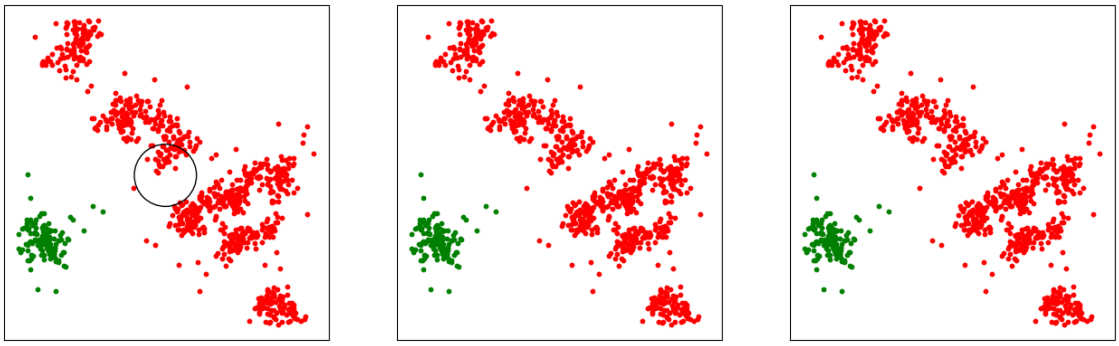
Nella presente sezione, andiamo ad analizzare i tempi di esecuzione dell'algoritmo approssimato e li confrontiamo con quelli ottenuti dall'algoritmo esatto. In particolare, andremo a variare, uno per volta, i parametri n, ε, $MinPts$ e ρ e osserveremo la relazione tra essi e il tempo di esecuzione. Gli esperimenti sono stati eseguiti utilizzando come input dei dataset sintetici, generati dall'algoritmo citato nella sezione precedente, per valori di dimensionalità pari a $3, 5, 7$. Successivamente, si farà riferimento a questi dataset con la seguente nomenclatura: *Gen-3D, Gen-5D, Gen-7D*. Inoltre, è stato preso in considera-



(a) DBSCAN Esatto, $\varepsilon = 50$ **(b)** Approssimato, $\varepsilon = 50, \rho = 1$ **(c)** Approssimato, $\varepsilon = 50, \rho = 0.1$



(d) DBSCAN Esatto, $\varepsilon = 100$ **(e)** Approssimato, $\varepsilon = 100, \rho = 1$ **(f)** Approssimato, $\varepsilon = 100, \rho = 0.1$



(g) DBSCAN Esatto, $\varepsilon = 150$ **(h)** Approssimato, $\varepsilon = 150, \rho = 1$ **(i)** Approssimato, $\varepsilon = 150, \rho = 0.1$

Figura 5.1: Nella prima colonna i risultati di clustering dell'algoritmo esatto, nella seconda e terza colonna i cluster restituiti dall'algoritmo approssimato, al variare del parametro ε.

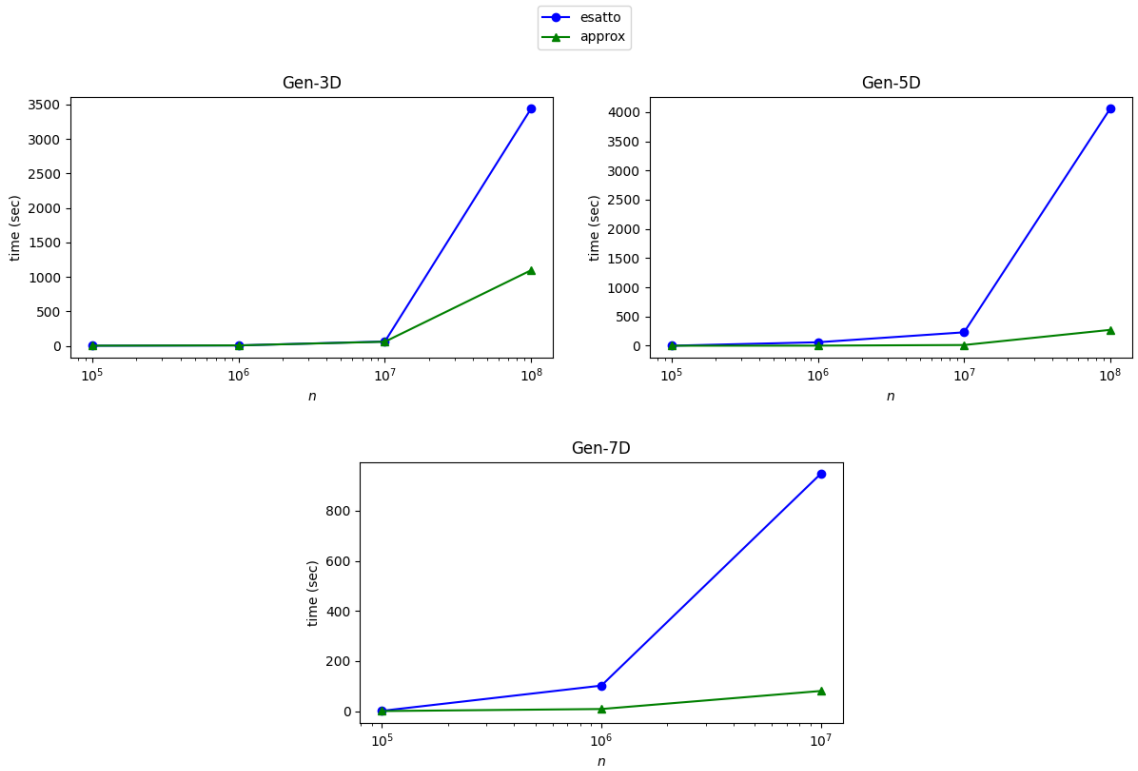


Figura 5.2: Confronto tra il tempo di esecuzione e il parametro n.

zione anche un dataset reale, denominato *PAMAP2*, caratterizzato da una cardinalità di $3,850,505$ punti e una dimensionalità pari a 4, fornito da Gan e Tao [7].

5.2.1 Variazione del parametro n

Nella seguente analisi andremo a confrontare i tempi di esecuzione dell'algoritmo approssimato con quelli dell'algoritmo esatto, al variare del parametro n, per valori a partire da $100k$ fino ad arrivare a $100m$. I restanti parametri sono fissati come segue: $\varepsilon = 5000$, $MinPts = 100$ e $\rho = 0.001$. I risultati ottenuti sono illustrati nei grafici della Figura 5.2, si noti che la scala nell'asse delle ascisse è logaritmica.

Osservando il primo grafico, relativo alla dimensionalità 3, si può osservare che per valori del parametro n fino a $10m$, la differenza tra i due algoritmi è minima, è invece notevole per $n = 100m$, dove l'algoritmo approssimato impiega circa un quarto del tempo rispetto a quello esatto. Una situazione simile si presenta nel secondo grafico, relativo alla dimensionalità 5, dove la discrepanza tra i due algoritmi inizia per $n = 10m$ e diventa piuttosto evidente al valore successivo. Per quanto riguarda l'ultimo grafico, relativo alla dimensionalità 7, si noti innanzitutto la mancanza dei risultati relativi al valore $n = 100m$, i quali

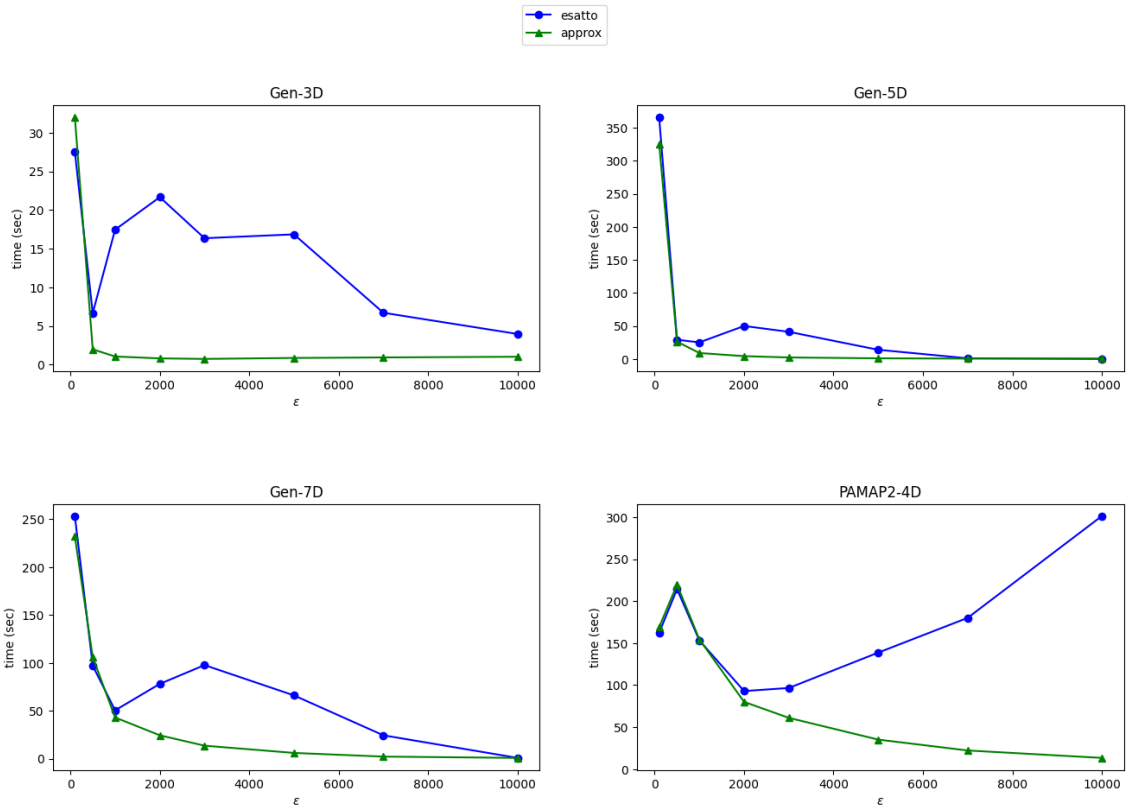


Figura 5.3: Confronto tra il tempo di esecuzione e il parametro ε.

superavano le sei ore e non sono stati misurati. Tuttavia, in questo caso il divario tra i due algoritmi si può notare a partire da valori inferiori, ovvero $n = 1m$ e diventa considerevole per $n = 10m$.

Possiamo concludere che la differenza, per quanto riguarda il tempo di esecuzione, tra l'algoritmo approssimato e quello esatto inizia ad essere notevole per cardinalità dei dataset a partire dall'ordine del milione.

5.2.2 Variazione del parametro ε

Andiamo ora ad analizzare l'influenza del parametro ε sui tempi di esecuzione dei due algoritmi. Sono stati presi in considerazione valori del parametro a partire da $\varepsilon = 100$ fino a $\varepsilon = 10k$; i rimanenti parametri sono stati fissati come segue: $n = 1m$, $MinPts = 100$, $\rho = 0.001$. I risultati ottenuti sono riportati nei grafici della Figura 5.3.

In generale, nei quattro grafici si può notare come l'algoritmo approssimato sia più veloce dell'algoritmo esatto, in alcuni casi in modo più considerevole in altri meno. Inoltre, l'algoritmo approssimato è piuttosto efficiente per un ampio range di valori del parametro

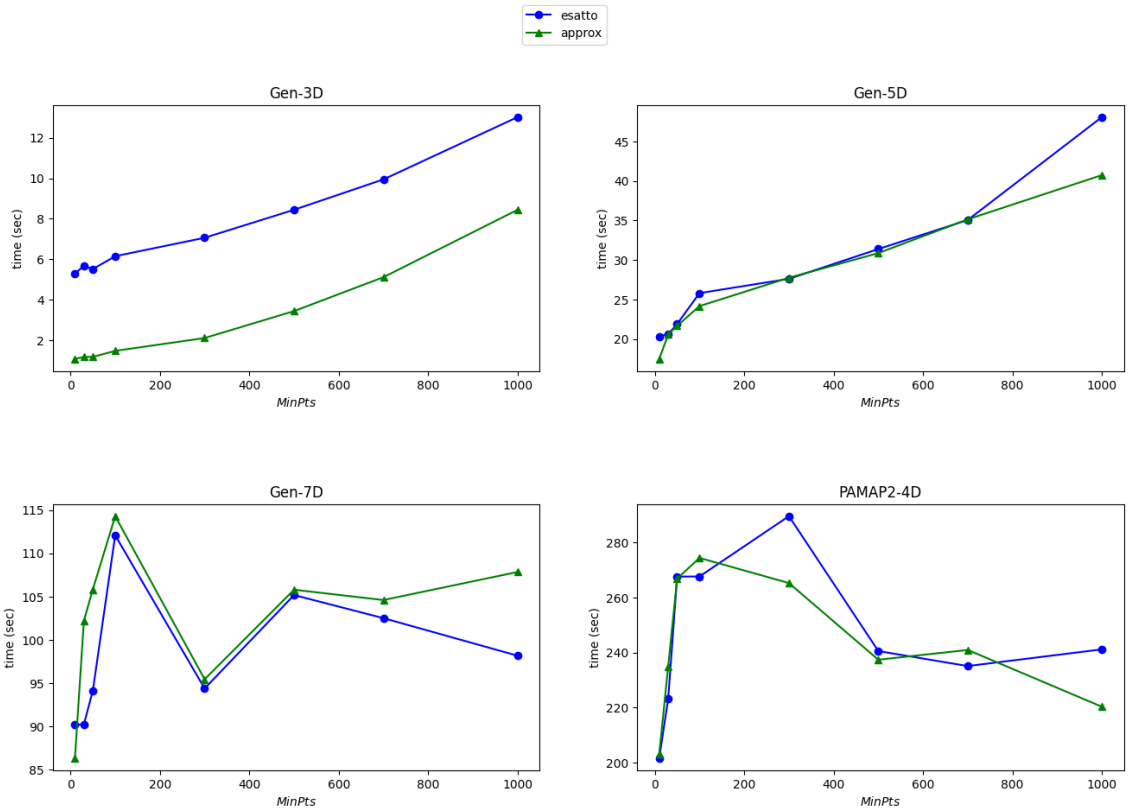


Figura 5.4: Confronto tra il tempo di esecuzione e il parametro $MinPts$.

ε. Questa proprietà è importante per andare a decidere il valore di ε più adatto per un determinato dataset, in quanto è possibile provare un ampio intervallo di valori, essendo che il tempo di esecuzione non risulta essere un problema.

5.2.3 Variazione del parametro $MinPts$

Passiamo ora ad esaminare la relazione tra il parametro $MinPts$ e il tempo di esecuzione dei due algoritmi. Le misurazioni sono state eseguite per valori del parametro a partire da $MinPts = 10$ fino a $MinPts = 1000$; i restanti parametri sono stati fissati come segue: $n = 1m, \varepsilon = 500, \rho = 0.001$. La Figura 5.4 mostra i risultati ottenuti da tali esperimenti.

Osservando i grafici in figura si può notare che al variare del parametro $MinPts$, i tempi di esecuzione si discostano di pochi secondi. Possiamo quindi dedurre che il valore del parametro $MinPts$ non influenza particolarmente il tempo di esecuzione degli algoritmi.

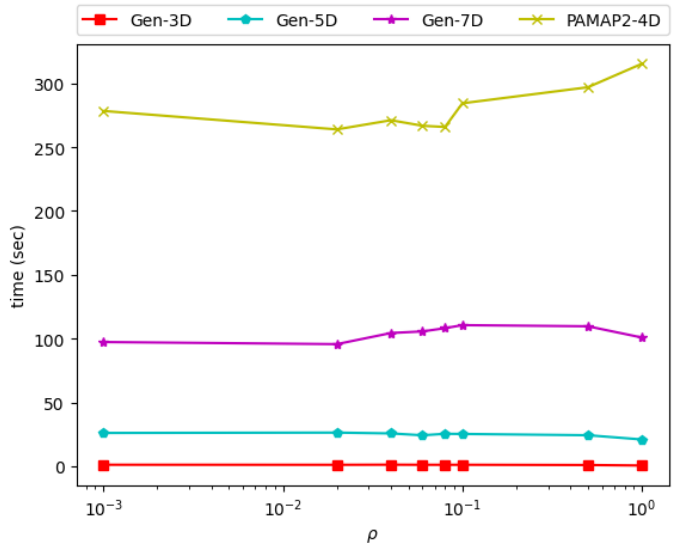


Figura 5.5: Confronto tra il tempo di esecuzione e il parametro ρ.

5.2.4 Variazione del parametro ρ

Infine, prendiamo in considerazione la variazione del parametro ρ relativo soltanto all'algoritmo approssimato. Gli esperimenti sono stati effettuati per valori del parametro a partire da $\rho = 0.001$ fino a $\rho = 1$; i rimanenti parametri sono stati fissati come segue: $n = 1m, \varepsilon = 500, MinPts = 100$. I risultati ottenuti sono descritti nel grafico della Figura 5.5, che presenta una scala logaritmica sull'asse delle ascisse.

A giudicare dal grafico, anche il parametro ρ non influenza in modo considerevole il tempo di esecuzione dell'algoritmo approssimato. Questo potrebbe risultare strano, considerato che la complessità temporale presentava una costante nascosta, pari a $(1/\rho)^{d-1}$, dovuta al costo di un'approximate range count query. Tuttavia, tale costo è relativo al caso peggiore, che non sempre si verifica, infatti spesso l'algoritmo che esegue le query termina in anticipo, senza dover visitare l'albero completo.

5.3 Suddivisione del tempo di esecuzione

In quest'ultima sezione, analizzeremo la distribuzione del tempo totale di esecuzione in relazione alle quattro procedure in cui si suddivide l'algoritmo, presentate nella Sezione 3.4. Per condurre questo esperimento, abbiamo configurato i parametri nel seguente modo: $\varepsilon = 500, MinPts = 100, \rho = 0.001$.

$n = 1m$	Gen-3D	%	Gen-5D	%	Gen-7D	%
Costruzione della griglia	0,10s	8,08%	0,15s	0,59%	0,29s	0,34%
Determinare i core point	0,18s	14,93%	10,77s	42,05%	43,07s	50,38%
Creazione del grafo G e determinare i cluster	0,92s	76,96%	14,69s	57,34%	42,09s	49,24%
Clustering dei border point	0,0003s	0,02%	0,01s	0,02%	0,03s	0,04%
Tempo totale di esecuzione	1,20s	100,00%	25,61s	100,00%	85,49s	100,00%

Tabella 5.1: Distribuzione del tempo totale di esecuzione ($n = 1m$).

$n = 10m$	Gen-3D	%	Gen-5D	%	Gen-7D	%
Costruzione della griglia	1,67s	9,67%	2,28s	0,79%	3,39s	0,31%
Determinare i core point	2,68s	15,53%	128,90s	44,58%	500,27s	45,78%
Creazione del grafo G e determinare i cluster	12,91s	74,73%	157,87s	54,59%	588,69s	53,87%
Clustering dei border point	0,01s	0,08%	0,12s	0,04%	0,48s	0,04%
Tempo totale di esecuzione	17,27s	100,00%	289,17s	100,00%	1092,83s	100,00%

Tabella 5.2: Distribuzione del tempo totale di esecuzione ($n = 10m$).

Nella Tabella 5.1 sono dettagliate le misurazioni relative a tre dataset sintetici con cardinalità $n = 1m$ e dimensionalità rispettivamente uguale a $3, 5, 7$. Lo stesso vale per la Tabella 5.2 con la cardinalità pari a $n = 10m$ e infine nella Tabella 5.3 sono riportati i risultati del dataset reale PAMAP2-D4 avente cardinalità $n = 3,850,505$.

È evidente che, nel contesto dei dataset sintetici, le procedure che emergono chiaramente come predominanti in termini di tempo impiegato sono quelle relative all'individuazione dei core point e alla creazione del grafo G, che include anche l'assegnazione dei cluster. Nel caso del dataset reale, invece, le procedure che prevalgono sulle altre, sono quella volta a determinare i core point e quella che si occupa del clustering dei border point. La variazione dei tempi di esecuzione delle procedure tra i dataset sintetici e quello reale è

$n = 3,850,505$	PAMAP2-D4	%
Costruzione della griglia	2,34s	0,98%
Determinare i core point	111,45s	46,69%
Creazione del grafo G e determinare i cluster	8,86s	3,71%
Clustering dei border point	116,03s	48,61%
Tempo totale di esecuzione	238,67s	100,00%

Tabella 5.3: Distribuzione del tempo totale di esecuzione.

dovuta al fatto che in quest'ultimo l'algoritmo, configurato con i parametri indicati, individua un alto numero di cluster. Inoltre, da misurazioni più dettagliate, è emerso che la maggior parte del tempo impiegato dalle due procedure, che si occupano di determinare i core point e costruire il grafo G, è dedicato a ottenere la lista delle celle ε-vicine.

Ottimizzazioni. Vediamo alcune ottimizzazioni volte a migliorare i tempi di esecuzione delle procedure predominanti. Per rendere più efficiente l'operazione di accesso alle celle ε-vicine ad una data cella, si possono salvare tutte le celle all'interno di un R-Tree, in sostituzione all'hash_map. Tale struttura dati è in grado di indicizzare spazi multidimensionali in modo efficiente. Inoltre, una volta calcolate le celle ε-vicine ad una cella c, queste possono essere salvate in una lista, in modo tale da non essere ricalcolate successivamente. Per quanto riguarda, invece, la procedura relativa alla creazione del grafo G può essere ottimizzata facendo ricorso alla struttura dati disjoint-set [3]. Un approccio di questo tipo permette di diminuire considerevolmente il numero di controlli sull'esistenza di un arco, in quanto è evitabile se due celle appartengono già alla medesima componente connessa.

Capitolo 6

Conclusioni

L'approccio DBSCAN è una valida tecnica per il clustering basato sulla densità ed è ampiamente utilizzato in ambiti come il data mining e il machine learning. Tuttavia, per dimensioni superiori a due, la complessità temporale del DBSCAN al caso peggiore è dimostrata essere superlineare, la qual cosa lo rende incompatibile nella gestione di grandi moli di dati. Vista l'impossibilità nel realizzare un algoritmo con complessità temporale quasi lineare, ne è stata progettata una versione approssimata.

Nel presente lavoro di tesi, si è andato a studiare l'algoritmo ρ-approximate DBSCAN, che assicura una buona qualità nell'approssimazione dei cluster e risulta efficiente dal punto di vista computazionale. Inoltre, è stata eseguita un'analisi delle prestazioni relative ad un'implementazione di tale algoritmo, dalla quale si è potuto evincere che tale algoritmo approssimato è piuttosto veloce per un ampio range di valori assunti dai parametri. Grazie a questa proprietà l'algoritmo può essere impiegato allo scopo di determinare valori significativi dei parametri per un determinato dataset. Esso consente inoltre di svolgere un'analisi rigorosa dei cluster, potendo spaziare su un vasto range di valori, mantenendo un tempo di esecuzione ragionevole. Possiamo quindi concludere che un ulteriore scopo dell'algoritmo approssimato consiste nel filtrare i valori assunti dai parametri, che verranno successivamente impiegati nell'esecuzione dell'algoritmo esatto.

Bibliografia

[1] Pankaj K. Agarwal, Herbert Edelsbrunner e Otfried Schwarzkopf. "Euclidean Minimum Spanning Trees and Bichromatic Closest Pairs". In: *Discret. Comput. Geom.* 6 (1991), pp. 407–422. DOI: 10.1007/BF02574698. URL: https://doi.org/10.1007/BF02574698.

[2] Sunil Arya e David M. Mount. "Approximate Range Searching". In: *Comput. Geom. Theory Appl.* 17.3–4 (dic. 2000), pp. 135–152. ISSN: 0925-7721. DOI: 10.1016/S0925-7721(00)00022-5. URL: https://doi.org/10.1016/S0925-7721(00)00022-5.

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[4] Jeff Erickson. "On the Relative Complexities of Some Geometric Problems". In: *Proc. 7th Canad. Conf. Comput. Geom.* (nov. 1995).

[5] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.

[6] Junhao Gan e Yufei Tao. "On the hardness and approximation of euclidean DBSCAN". In: *ACM Transactions on Database Systems* 42 (lug. 2017), pp. 1–45. DOI: 10.1145/3083897.

[7] Junhao Gan e Yufei Tao. *Source code and binary of several DBSCAN implementations*. Available on line. 2015. URL: https://sites.google.com/view/approxdbscan/old-versions.

[8] Ade Gunawan. “A faster algorithm for DBSCAN”. In: 2013. URL: https://api.semanticscholar.org/CorpusID:61583059.

[9] Jirí Matousek. “Range Searching with Efficient Hiearchical Cutting”. In: *Discret. Comput. Geom.* 10 (1993), pp. 157–182. DOI: 10.1007/BF02573972. URL: https://doi.org/10.1007/BF02573972.

[10] Hermann Tropf e H. Herzog. “Multimensional Range Search in Dynamically Balanced Trees”. In: *Angew. Inform.* 23 (1981), pp. 71–77. URL: https://api.semanticscholar.org/CorpusID:26857103.